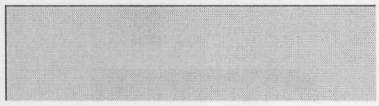

EE Expert Robert Ashby

Embedded Engineering

Archive | Main EE Expert | Guides & Experts

Multiplication and Division Made Easy

Multiplication and division might seem like easy things to do. However, you often have microcontrollers/microprocessors that don't have built-in multiply and divide commands. It is then up to the programmer to write macros or subroutines to handle more complex math. Doesn't sound too hard.

It's really not too hard to write a routine to multiply or divide. It can be difficult, however, to write good multiply and divide routines. Some of the characteristics of good routines are that they be short, concise, and consistently use as little memory as possible. In talking with students and other professionals, I ask them how would they write multiply and divide routines using add, subtract, and other basic programming commands. I often get the same method that I first came up with when I tackled the problem.

We want to multiply two numbers A*B
Result = 0
If (B = 0) Then Exit
Result = Result + A
B = B -1
If (B = 0) Then Exit Else GOTO 3

We want to divide two numbers A/B
Result = 0
Remainder = A
If (B < A) Then Exit
Remainder = Remainder - B
Result = Result + 1
GOTO 3

These routines work and they have some advantages: you use very little RAM (i.e., code space), and the routines are very straightforward and easy to follow. However, there is one disadvantage--the routines could take a very long time to execute.

For example, the multiplication routine executes quickly if B=3. If B=5,000, the routine takes much longer. The divide routine runs into the same problem as the ratio of A to B becomes very large. Anyone who spends their days trying to squeeze performance out of the bits and bytes world knows that this is a no-no. Routines like this would cause you to spend all your time trying to find out why the chip resets when the watchdog timers expire

while a big number gets processed.

Fortunately, there is a better way. I was shown the following methods and pass them on to you as useful tools. There isn't a great secret, you just need to get out of that old mundane base 10 world and think like a computer.

The binary world has one reoccurring advantage -- when you shift numbers to the left once, you multiply that number by 2. If you shift numbers right once, you divide by two. Not too hard, right? After all we've followed a similar rule since we were little in our decimal world. Shift one digit to the left and we multiply by ten, shift one digit to the right and we divide by ten.

Using this simple rule with addition and subtraction, we can now write multiply and divide routines that are accurate, expandable, use very little code or RAM, and take approximately the same amount of cycles no matter what the numbers are. My examples below are byte-sized for simplicity, but the same pattern can be used on operands of any size. You just need the register space available to expand on this.

Multiplication
Lets start with two numbers, A * B. For this example, I will say that A = 11 and B = 5.

In binary, A = 00001011 and B = 0000101.

When multiplying two byte-sized numbers, I know that the result can always be expressed in two bytes, therefore, RESULT is word-sized, and TEMP is word-sized. COUNT needs only to be one byte.

1. RESULT = 0	This is where the answer will end up
2. TEMP = A	Necessary to have a word-sized equivalent for shifting
3. COUNT = 8	This is because we are multiplying by an 8-bit number
4. Shift B right through carry	Find out if the lowest bit is 1
5. If (carry = 1) then RESULT = RESULT + TEMP	
6. TEMP = TEMP + TEMP ;Multiply TEMP * 2 to set up for next loop	
7. COUNT = COUNT ? 1	
8. If (COUNTER = 0) then exit else GOTO 4	

Look at the mechanics of this. As I rotate or shift B through carry each time, I am basically moving left in B each time through the loop, and then deciding whether B has a 1 or a 0 in that location (remember moving left is multiplying by two). At the same time, I am shifting TEMP left each time since the binary digit I am checking in B is double the magnitude as the previous time through the loop. Then all that is left to do is add the TEMP value if the value of the binary digit in B is 1, or not add it if B has a 0 in that location. By the time COUNT = 0, you have the final result in

RESULT. The loop works the same no matter how large your numbers are. The subroutine has a somewhat small range of possible machine cycles that it takes and still remains compact and uses a minimal amount of RAM.

Lets Look at our example problem each time it reaches step #8. (Note x = Don't care.)

Loop count	RESULT	B	TEMP	COUNT
1	00000000 00001011	x0000010	00000000 00010110	7
2	00000000 00001011	xx000001	00000000 00101100	6
3	00000000 00110111	xxx00000	00000000 01011000	5
4	00000000 00110111	xxxx0000	00000000 10110000	4
5	00000000 00110111	xxxxx000	00000001 01100000	3
6	00000000 00110111	xxxxxx00	00000010 11000000	2
7	00000000 00110111	xxxxxxx0	00000101 10000000	1
8	00000000 00110111	xxxxxxxx	00001011 00000000	0

Division
Now that Multiplication is clear, division is just multiplication in reverse. Let's take the numbers A = 102 and B = 20 and perform A/B.

In binary: A = 01100110 B = 00010100

Since I am dealing with integers, I know that A/B has a RESULT less than or equal to A .
Therefore RESULT is one byte, and REMAINDER is one byte. TEMP is two bytes.

1. RESULT = 0 This is where the answer will end up
2. REMAINDER = 0 Accomplished by loading the
3. COUNT = 8 This is because we are dividing by an 8 bit number

4. RESULT = RESULT + RESULT
5. Shift A left through carry
6. Shift REMAINDER left through carry
7. If REMAINDER >= B Then RESULT = RESULT+1 and REMAINDER = REMAINDER - B
8. COUNT = COUNT - 1
9. If (COUNTER = 0) then exit else GOTO 4

This might seem somewhat foreign, but it's really the same type of division that you've always known. Look at how many digits in the top part of A I have to include before B will divide into those digits. Once I have a number I subtract that division and then continue. Follow through the table with our example numbers and see if it becomes clear.

Lets Look at our example problem each time it reaches step #8. Note x = Don't care

Loop Count	A	RESULT	REMAINDER	COUNT
1	1100110x	00000000	00000000	7
2	100110xx	00000000	00000001	6
3	00110xxx	00000000	00000011	5
4	0110xxxx	00000000	00000110	4
5	110xxxxx	00000000	00001100	3
6	10xxxxxx	00000001	00000101	2
7	0xxxxxxx	00000010	00001011	1
8	xxxxxxxx	00000101	00000010	0

IT WORKS!

There always seem to be several ways to do things, and I would never say to you that these are the best math routines for all situations. However, they are very flexible and easy to use. The can easily be adapted for 16 bit, 32 bit, 64 bit, or higher math and still work just as well.

The time that it takes for the math to execute depends on the size of the operands in bits, not the actual value of the operands, giving you more or less consistent time for the routine, a very desirable trait.

Embedded Engineering